

Course Review

Final Exam

- ✦ 20/5/2016, Friday 8:30AM - 11:30AM, Tsang Shiu Tim Art Hall
- ✦ Include all topics but with more emphasis on materials after the midterm.
- ✦ Similar to midterm.
- ✦ Closed book with a focus on concepts and principles.

Final Exam

- ✦ Basic OOP (20%)
 - Polymorphism
 - Object initialization
 - Abstract class, Interface, and Inner class
- ✦ Exception Handling(10%)
- ✦ Generics (25%)
- ✦ Multithreading (25%)
- ✦ Others (strings, networking, file I/O, GUI) (20%)

Java

- ✦ Object oriented programming
 - No explicit pointers but objects are references
 - Extends one class but can implement multiple interfaces.
- ✦ Exception handling is the norm: must handle all checked exceptions.
- ✦ Generics.
- ✦ Threads.
- ✦ Comprehensive APIs for dealing with GUI, networking, ...

Generics

Casting from Superclass to Subclass

Explicit casting must be used when casting an object from a superclass to a subclass. This type of casting may not always succeed.

```
Apple x = (Apple) fruit;
```

```
Orange x = (Orange) fruit;
```

(From Inheritance Lecture)

Casting is bad

- ✦ Remember the devil: `void*`?
- ✦ If: `Object o = new Object()` and `Apple a = (Apple)o`, you get a runtime error.
- ✦ It was mostly used when you use collection classes such as “LinkedList”.
- ✦ No longer necessary with the use of Generics

(From Inheritance Lecture)

What is Generics?

- ✦ *Generics* is the capability to parameterize types.
- ✦ You can define a class or a method with generic types that can be substituted using concrete types by the compiler.
- ✦ For example, you may define a generic stack class that stores the elements of a generic type. From this generic class, you may create a stack object for holding strings and a stack object for holding numbers. Here, strings and numbers are concrete types that replace the generic type.

Why Generics?

- ✦ The key benefit of generics
 - Enable errors to be detected at compile time rather than at runtime.
 - A generic class or method permits you to specify allowable types of objects that the class or method may work with.
 - If you attempt to use the class or method with an incompatible object, the compile error occurs.

Generic Type

```
package java.lang;

public interface Comparable {
    public int compareTo(Object o)
}
```

(a) Prior to JDK 1.5

```
package java.lang;

public interface Comparable<T> {
    public int compareTo(T o)
}
```

(b) JDK 1.5

Runtime error

```
Comparable c = new Date();
System.out.println(c.compareTo("red"));
```

(a) Prior to JDK 1.5

```
Comparable<Date> c = new Date();
System.out.println(c.compareTo("red"));
```

(b) JDK 1.5

Generic Instantiation

Improves reliability

Compile error

Generic Motivation

_interfaces.ComparableRectangle, ComparableRectangleWithGeneric

Generic ArrayList in JDK 1.5

java.util.ArrayList

```
+ArrayList()
+add(o: Object) : void
+add(index: int, o: Object) : void
+clear(): void
+contains(o: Object): boolean
+get(index: int) : Object
+indexOf(o: Object) : int
+isEmpty(): boolean
+lastIndexOf(o: Object) : int
+remove(o: Object): boolean
+size(): int
+remove(index: int) : boolean
+set(index: int, o: Object) : Object
```

(a) ArrayList before JDK 1.5

java.util.ArrayList<E>

```
+ArrayList()
+add(o: E) : void
+add(index: int, o: E) : void
+clear(): void
+contains(o: Object): boolean
+get(index: int) : E
+indexOf(o: Object) : int
+isEmpty(): boolean
+lastIndexOf(o: Object) : int
+remove(o: Object): boolean
+size(): int
+remove(index: int) : boolean
+set(index: int, o: E) : E
```

(b) ArrayList in JDK 1.5

No Casting Needed

```
ArrayList<Double> list = new ArrayList<Double>();
```

```
list.add(5.5); // 5.5 is automatically converted to new Double(5.5)
```

```
list.add(3.0); // 3.0 is automatically converted to new Double(3.0)
```

```
Double doubleObject = list.get(0); // No casting is needed
```

```
double d = list.get(1); // Automatically converted to double
```


Generic Types Restriction

- ✦ Generic types must be reference types.

- ✦ The following statement is wrong:

```
ArrayList<int> ints = new ArrayList<int>();
```

- ✦ Use wrapper class types:

```
ArrayList<Integer> ints = ...
```

Multiple Type Parameter

✦ Our coffee shop lunch menu

- Meat + Starch + Soup
- (Pork/Beef/Chicken)+(Rice/Italy noodle/Fries)+(Cream/Vegetable soup)

```
class Lunch<M,St,S> {  
    M meat_  
    St starch_  
    S soup_  
    public Lunch(M meat, St starch, S soup){  
        meat_ = meat;  
        starch_=starch;  
        soup_=soup;  
    }  
}
```

```
Beef b = new Beef();  
Rice r = new Rice();  
MushroomSoup s = new MushroomSoup();  
Lunch<Beef, Rice, MushroomSoup> l = new Lunch<Beef,  
Rice, MushroomSoup>(b,r,s);
```

Problem 1:

Incorrect Parameter Types

```
Node b = new Node();  
TreeNode r = new TreeNode();  
BTreeNode s = new BTreeNode();  
Lunch<Node, TreeNode, BTreeNode> l = new  
Lunch<Node, TreeNode, BTreeNode>(b,r,s);
```

We want to more properly say the following:

```
Lunch<subtypes of Meat, subtypes of Starch, subtypes of Soup>
```

Bounded Generic Types

```
public class Meat { ... }
public class Starch { ... }
public class Soup { ... }
public class Beef extends Meat { ... }
public class CreamSoup extends Soup { ... }

class Lunch <M extends Meat, St extends Starch, S extends Soup>
{
    M meat_;
    St starch_;
    S soup_;
    public Lunch(M meat, St starch, S soup){
        meat_ = meat;
        starch_=starch;
        soup_=soup;
    }
}
```

Generic Methods

```
public static <E> void print(E[] list) {  
    for (int i = 0; i < list.length; i++)  
        System.out.print(list[i] + " ");  
    System.out.println();  
}
```

```
public static void print(Object[] list) {  
    for (int i = 0; i < list.length; i++)  
        System.out.print(list[i] + " ");  
    System.out.println();  
}
```

```

public class Test {
    public static void main(String[] args) {
        Integer[] integers = {1,2,3,4,5};
        String[] strings = {"London", "Paris", "New York", "Austin"};
        Object[] objects = {"London", "Paris", "New York", "Austin", 1};
        Test.<Integer>print(integers);
        Test.<String>print(strings);
        Test.print(objects);
        Test.print1(integers);
        Test.print1(strings);
        Test.print1(objects);
    }
    public static <E> void print(E[] list) {
        for (int i = 0; i < list.length; i++)
            System.out.print(list[i] + " ");
        System.out.println();
    }
    public static void print1(Object[] list) {
        for (int i = 0; i < list.length; i++)
            System.out.print(list[i] + " ");
        System.out.println();
    }
}

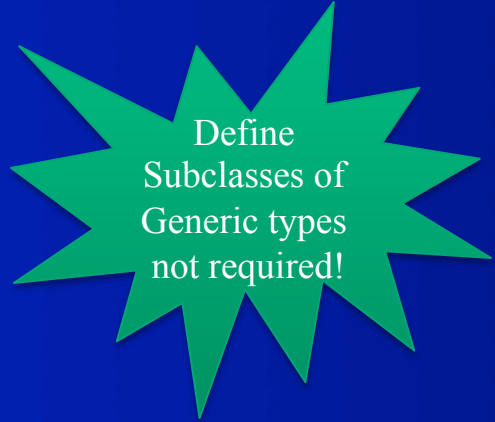
```

GenericTest

Problem 2:

Inheritance of Generic Types

- ✦ ClassA extends ClassB
- ✦ Subtype
 - ClassA is a subtype of ClassB
 - Stack<ClassA> is NOT a subtype of Stack<ClassB>
- ✦ Are we losing polymorphism?
 - This creates trouble for us
 - Customer requests a refund because they want more flexibility





Define
Subclasses of
Generic types
not required!

Are we losing polymorphism?

```
class Stack<T>{  
    public void pushAll(Stack<T> s){  
        for(//there is element in me)  
            s.push(pop());  
    }  
    public void popAll(Stack<T> s) {  
        for(//there is element in s)  
            push(s.pop());  
    }  
}
```



```
//Let us say ClassB is a  
//subclass of ClassA  
Stack<ClassA> s1 = new Stack<ClassA>();  
  
Stack<ClassB> s2 = new Stack<ClassB>();  
  
s2.pushAll(s1);   
//should be OK since ClassB is also a  
//type of ClassA. s1 that handles the  
//super type can handle subtypes  
  
s1.popAll(s2);   
//should be OK since ClassA is a super  
//type of ClassB. Again, s1 should be  
//able to take subtypes of ClassA.
```

It seems that parameter s should behave polymorphically

We need “wild card”

- ✦ In defining a generic type
 - “E extends ClassA” represents all subtypes of ClassA
- ✦ In defining a container variable that behaves polymorphically
 - “? extends” represent any sub types.
 - “? super” represent any super types

Support Polymorphism



Versi
on 4

```
class Stack<T>{  
    public void pushAll(Stack<? super T> s){  
        for(//there is element in me)  
            s.push(pop());  
    }  
    public void popAll(Stack<? extends T> s) {  
        for(//there is element in s)  
            push(s.pop());  
    }  
}
```

```
Stack<ClassA> s1 = new Stack<ClassA>();  
Stack<ClassB> s2 = new Stack<ClassB>();  
s2.pushAll(s1);  
s1.popAll(s2);
```

Wildcards

Why wildcards are necessary? One more example.

WildCardDemo1

| | |
|-------------|-----------------------|
| ? | unbounded wildcard |
| ? extends T | bounded wildcard |
| ? super T | lower bound wildcard: |

WildCardDemo2

WildCardDemo3

Erasure on Generics

Generics are implemented using an approach called *type erasure*.

```
public static <T> int count(T[] anArray, T elem) {  
    int cnt = 0;  
    for (T e : anArray)  
        if (e.equals(elem))  
            ++cnt;  
    return cnt;  
}
```

```
public static int count(Object[] anArray, Object elem) {  
    int cnt = 0;  
    for (Object e : anArray)  
        if (e.equals(elem))  
            ++cnt;  
    return cnt;  
}
```

Erasure on Generics

- ✦ Generics are a wrapper over `java.lang.Object` to ensure backwards compatibility
- ✦ Generics are implemented using an approach called *type erasure*.
 - Unbound variable replaced by “Object”
 - Bounded variable “T extends ClassA”, “? super ClassB” replaced by bound, “ClassA” or “ClassB”.
- ✦ The types used to instantiate generics serve as “comments”.
- ✦ What generics really is:
 - Compile time checks
 - Hints to generate the casting correctly.
 - Enforcing the usage of the same type

Compile Time Checking

For example, the compiler checks whether generics is used correctly for the following code in (a) and translates it into the equivalent code in (b) for runtime use. The code in (b) uses the raw type.

```
ArrayList<String> list = new ArrayList<String>();  
list.add("Oklahoma");  
String state = list.get(0);
```

(a)

```
ArrayList list = new ArrayList();  
list.add("Oklahoma");  
String state = (String)(list.get(0));
```

(b)

Checks and Hints

```
class Parent<T> {  
    T compute(T t){return t;}  
}
```

```
p = new Parent<String>();  
String s = p.compute("s");  
String s = p.compute(circle);
```



```
class Parent{  
    Object compute(Object t){  
        return t;  }  
}
```

```
p = new Parent();  
String s = (String) p.compute("s");  
String s = (String) p.compute(circle);
```

Generics tells us p can only deal with Strings . This information is stored in the compiler not in the code

Type Enforcement

```
class Parent<T extends Geom> {  
    T compute(T t1, T t2){return t1.same(t2);}  
}
```

```
p = new Parent<Circle>();  
String s = p.compute(circle, triangle);
```

```
class Parent{  
    Geom compute(Geom t1, Geom t2){  
        return t1.same(t2);    }  
}
```

```
p = new Parent();  
Geom = (Circle) p.compute((Circle)circle, (Circle)triangle);
```

Generics tells us these two parameters have to be the same type, not just subclass of Geom. This information is stored in the compiler not in the code

Important Facts

It is important to note that a generic class is shared by all its instances regardless of its actual generic type.

```
GenericStack<String> stack1 = new GenericStack<String>();  
GenericStack<Integer> stack2 = new GenericStack<Integer>();
```

Although GenericStack<String> and GenericStack<Integer> are two types, but there is only one class GenericStack loaded into the JVM.

Restrictions on Generics

- ✦ Restriction 1: Cannot Create an Instance of a Generic Type. (i.e., `new E()`).
- ✦ Restriction 2: Generic Array Creation is Not Allowed. (i.e., `new E[100]`).
- ✦ Restriction 3: A Generic Type Parameter of a Class is Not Allowed in a Static Context.
- ✦ Restriction 4: Exception Classes Cannot be Generic.

Not allowed in a static context

```
public class Test<E> {  
    public static void m (E o1) { // Illegal  
    }  
    public static E o1; // Illegal  
    static {  
        E o2; // Illegal  
    }  
}
```

Not allowed in exception classes

```
public class MyException<T> extends Exception {}
```

```
try {  
    ...  
}  
catch (MyException <Integer> ex) {  
    ...  
}  
catch (MyException<Circle> ex) {  
    ...  
}
```

Threads

Creating Tasks and Threads

`java.lang.Runnable`

TaskClass



```
// Custom task class
public class TaskClass implements Runnable {
    ...
    public TaskClass(...) {
        ...
    }

    // Implement the run method in Runnable
    public void run() {
        // Tell system how to run custom thread
        ...
    }
    ...
}
```

```
// Client class
public class Client {
    ...
    public void someMethod() {
        ...
        // Create an instance of TaskClass
        TaskClass task = new TaskClass(...);

        // Create a thread
        Thread thread = new Thread(task);

        // Start a thread
        thread.start();
        ...
    }
    ...
}
```

run()

- ✦ The run() methods in a task class specifies how to perform the task. It's automatically invoked by JVM when a thread is started.
- ✦ You should not invoke it: doing so merely executes this method in the same thread; no new thread is started.

The Thread Class

«interface»
java.lang.Runnable



java.lang.Thread

| | |
|------------------------------------|---|
| +Thread() | Creates a default thread. |
| +Thread(task: Runnable) | Creates a thread for a specified task. |
| +start(): void | Starts the thread that causes the run() method to be invoked by the JVM. |
| +isAlive(): boolean | Tests whether the thread is currently running. |
| +setPriority(p: int): void | Sets priority p (ranging from 1 to 10) for this thread. |
| +join(): void | Waits for this thread to finish. |
| + <u>sleep(millis: long): void</u> | Puts the runnable object to sleep for a specified time in milliseconds. |
| + <u>yield(): void</u> | Causes this thread to temporarily pause and allow other threads to execute. |
| +interrupt(): void | Interrupts this thread. |

The Static sleep(milliseconds) Method

The sleep(long mills) method puts the thread to sleep for the specified time in milliseconds. For example, suppose you modify the code in Lines 53-57 in TaskThreadDemo.java as follows:

```
public void run() {  
    for (int i = 1; i <= lastNum; i++) {  
        System.out.print(" " + i);  
        try {  
            if (i >= 50) Thread.sleep(1);  
        }  
        catch (InterruptedException ex) {  
        }  
    }  
}
```

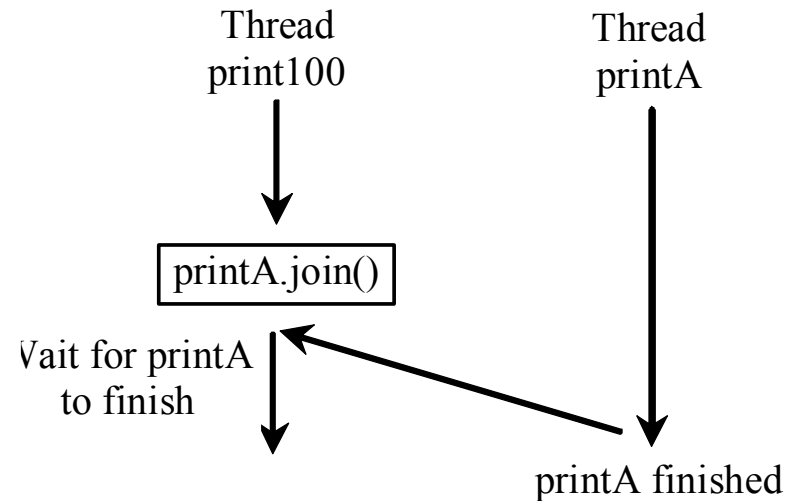
Every time a number (≥ 50) is printed, the print100 thread is put to sleep for 1 millisecond.

- ✦ The `sleep()` method may throw an `InterruptedException`, which is a checked exception.
- ✦ This rarely occurs, but you have to catch it as it is a checked exception.
- ✦ The same hold for the `join()` method.

The join() Method

You can use the join() method to force one thread to wait for another thread to finish. For example, suppose you modify the code in Lines 53-57 in TaskThreadDemo.java as follows:

```
public void run() {  
    Thread thread4 = new Thread(  
        new PrintChar('c', 40));  
    thread4.start();  
    try {  
        for (int i = 1; i <= lastNum; i++) {  
            System.out.print(" " + i);  
            if (i == 50) thread4.join();  
        }  
    }  
    catch (InterruptedException ex) {  
    }  
}
```



The numbers after 50 are printed after thread printA is finished.

Thread Priority

- ✦ Each thread is assigned a default priority of `Thread.NORM_PRIORITY`. You can reset the priority using `setPriority(int priority)`.

- ✦ Some constants for priorities include

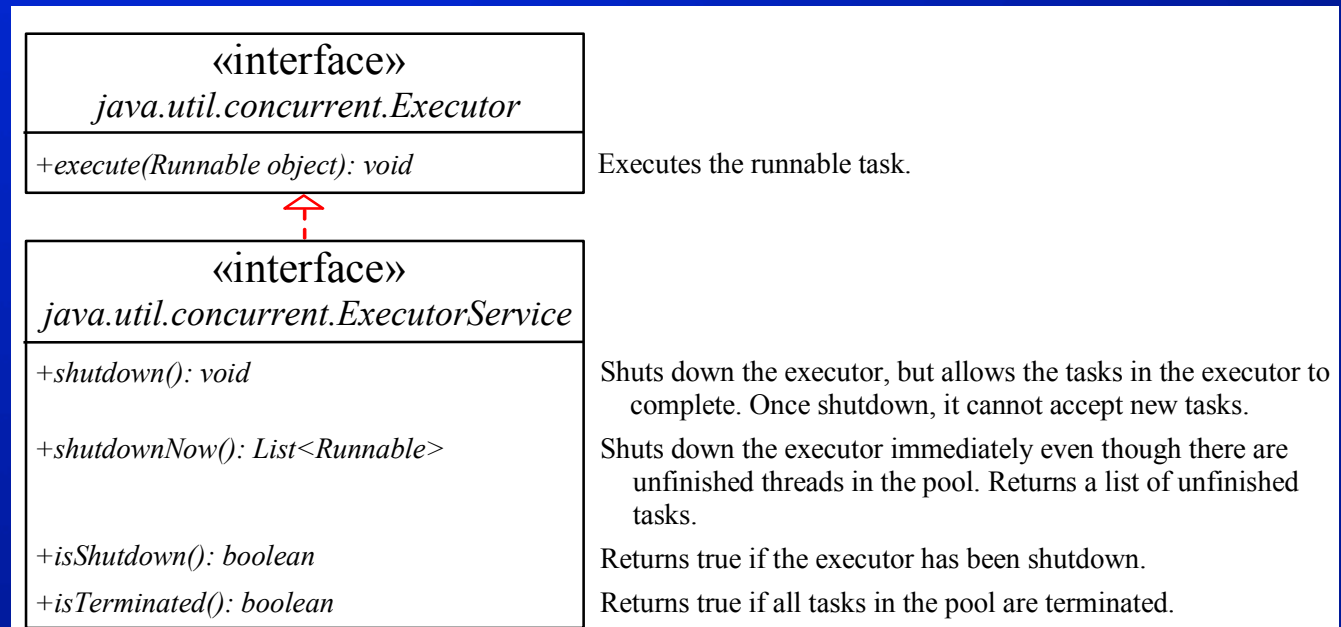
 - `Thread.MIN_PRIORITY`

 - `Thread.MAX_PRIORITY`

 - `Thread.NORM_PRIORITY`

Thread Pools

Starting a new thread for each task could limit throughput and cause poor performance. A thread pool is ideal to manage the number of tasks executing concurrently. JDK 1.5 uses the Executor interface for executing tasks in a thread pool and the ExecutorService interface for managing and controlling tasks. ExecutorService is a subinterface of Executor.



Creating Executors

To create an Executor object, use the static methods in the Executors class.

| java.util.concurrent.Executors |
|---|
| +newFixedThreadPool(numberOfThreads: int): ExecutorService |
| +newCachedThreadPool(): ExecutorService |

Creates a thread pool with a fixed number of threads executing concurrently. A thread may be reused to execute another task after its current task is finished.

Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.

[ExecutorDemo](#)

Thread Synchronization

A shared resource may be corrupted if it is accessed simultaneously by multiple threads. For example, two unsynchronized threads accessing the same bank account may cause conflict.

| Step | balance | thread[i] | thread[j] |
|------|---------|--|--|
| 1 | 0 | <code>newBalance = bank.getBalance() + 1;</code> | |
| 2 | 0 | | <code>newBalance = bank.getBalance() + 1;</code> |
| 3 | 1 | <code>bank.setBalance(newBalance);</code> | |
| 4 | 1 | | <code>bank.setBalance(newBalance);</code> |

Race Condition

What, then, caused the error in the example? Here is a possible scenario:

| Step | balance | Task 1 | Task 2 |
|------|---------|---------------------------|---------------------------|
| 1 | 0 | newBalance = balance + 1; | |
| 2 | 0 | | newBalance = balance + 1; |
| 3 | 1 | balance = newBalance; | |
| 4 | 1 | | balance = newBalance; |

The effect of this scenario is that Task 1 did nothing, because in Step 4 Task 2 overrides Task 1's result. Obviously, the problem is that Task 1 and Task 2 are accessing a common resource in a way that causes conflict. This is a common problem known as a *race condition* in multithreaded programs. A class is said to be *thread-safe* if an object of the class does not cause a race condition in the presence of multiple threads. As demonstrated in the preceding example, the Account class is not thread-safe.

The synchronized keyword

To avoid race conditions, more than one thread must be prevented from simultaneously entering certain part of the program, known as critical region. The critical region in the Listing 29.7 is the entire deposit method. You can use the synchronized keyword to synchronize the method so that only one thread can access the method at a time. There are several ways to correct the problem in Listing 29.7, one approach is to make Account thread-safe by adding the synchronized keyword in the deposit method in Line 45 as follows:

```
public synchronized void deposit(double amount)
```

Synchronizing Instance Methods and Static Methods

A synchronized method acquires a lock before it executes. In the case of an instance method, the lock is on the object for which the method was invoked. In the case of a static method, the lock is on the class. If one thread invokes a synchronized instance method (respectively, static method) on an object, the lock of that object (respectively, class) is acquired first, then the method is executed, and finally the lock is released. Another thread invoking the same method of that object (respectively, class) is blocked until the lock is released.

Synchronizing Statements

Invoking a synchronized instance method of an object acquires a lock on the object, and invoking a synchronized static method of a class acquires a lock on the class. A synchronized statement can be used to acquire a lock on any object, not just *this* object, when executing a block of the code in a method. This block is referred to as a *synchronized block*. The general form of a synchronized statement is as follows:

```
synchronized (expr) {  
    statements;  
}
```

The expression `expr` must evaluate to an object reference. If the object is already locked by another thread, the thread is blocked until the lock is released. When a lock is obtained on the object, the statements in the synchronized block are executed, and then the lock is released.

Synchronizing Statements vs. Methods

Any synchronized instance method can be converted into a synchronized statement. Suppose that the following is a synchronized instance method:

```
public synchronized void xMethod() {  
    // method body  
}
```

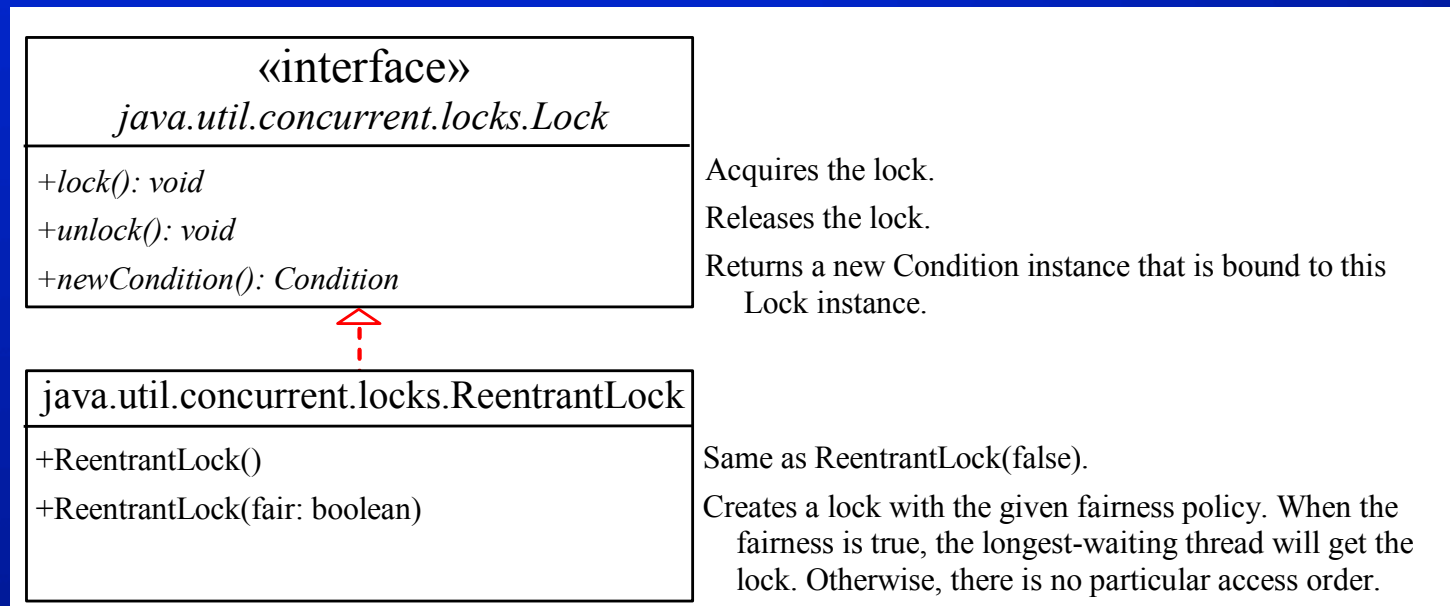
This method is equivalent to

```
public void xMethod() {  
    synchronized (this) {  
        // method body  
    }  
}
```

Synchronization Using Locks

A synchronized instance method implicitly acquires a lock on the instance before it executes the method.

JDK 1.5 enables you to use locks explicitly. The new locking features are flexible and give you more control for coordinating threads. A lock is an instance of the Lock interface, which declares the methods for acquiring and releasing locks, as shown in Figure 29.14. A lock may also use the newCondition() method to create any number of Condition objects, which can be used for thread communications.



Fairness Policy

ReentrantLock is a concrete implementation of Lock for creating mutual exclusive locks. You can create a lock with the specified fairness policy. True fairness policies guarantee the longest-wait thread to obtain the lock first. False fairness policies grant a lock to a waiting thread without any access order. Programs using fair locks accessed by many threads may have poor overall performance than those using the default setting, but have smaller variances in times to obtain locks and guarantee lack of starvation.

Example: Using Locks

This example revises `AccountWithoutSync.java` in Listing 29.7 to synchronize the account modification using explicit locks.

`AccountWithSyncUsingLock`

Cooperation Among Threads

The conditions can be used to facilitate communications among threads. A thread can specify what to do under a certain condition. Conditions are objects created by invoking the `newCondition()` method on a `Lock` object. Once a condition is created, you can use its `await()`, `signal()`, and `signalAll()` methods for thread communications, as shown in Figure 29.15. The `await()` method causes the current thread to wait until the condition is signaled. The `signal()` method wakes up one waiting thread, and the `signalAll()` method wakes all waiting threads.

«interface»

java.util.concurrent.Condition

+*await(): void*

+*signal(): void*

+*signalAll(): Condition*

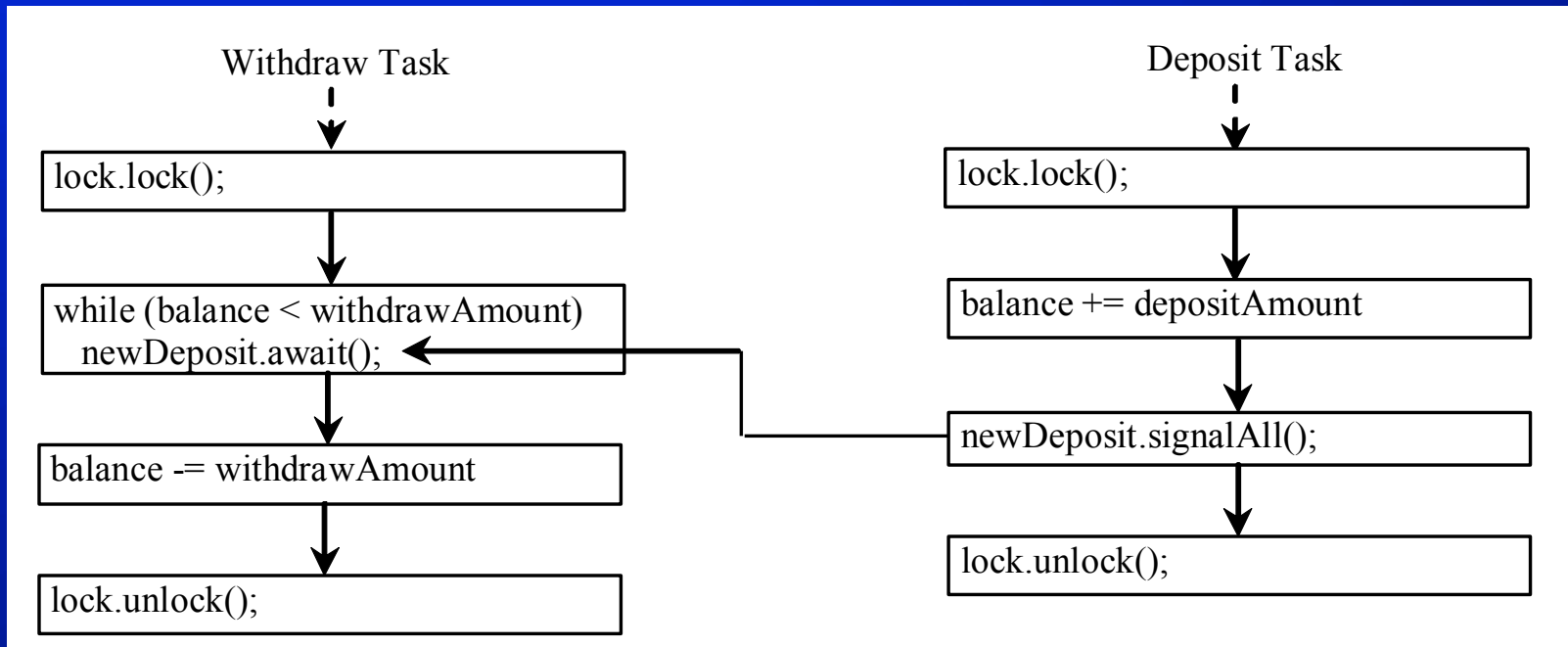
Causes the current thread to wait until the condition is signaled.

Wakes up one waiting thread.

Wakes up all waiting threads.

Cooperation Among Threads

To synchronize the operations, use a lock with a condition: newDeposit (i.e., new deposit added to the account). If the balance is less than the amount to be withdrawn, the withdraw task will wait for the newDeposit condition. When the deposit task adds money to the account, the task signals the waiting withdraw task to try again. The interaction between the two tasks is shown in Figure 29.16.



Example: Thread Cooperation

Write a program that demonstrates thread cooperation. Suppose that you create and launch two threads, one deposits to an account, and the other withdraws from the same account. The second thread has to wait if the amount to be withdrawn is more than the current balance in the account. Whenever new fund is deposited to the account, the first thread notifies the second thread to resume. If the amount is still not enough for a withdrawal, the second thread has to continue to wait for more fund in the account. Assume the initial balance is 0 and the amount to deposit and to withdraw is randomly generated.



```
Command Prompt
C:\book>java ThreadCooperation
Thread 1      Thread 2      Balance
Deposit 7           7
Deposit 1           8
Deposit 10          18
                Withdraw 9    9
                Withdraw 4    5
                Withdraw 3    2
Deposit 9           11
                Withdraw 5    6
                Withdraw 2    4
Deposit 3           7
```

ThreadCooperation

Java's Built-in Monitors

Locks and conditions are new in Java 5. Prior to Java 5, thread communications are programmed using object's built-in monitors. Locks and conditions are more powerful and flexible than the built-in monitor. However, it is easier to understand and use.

A *monitor* is an object with mutual exclusion and synchronization capabilities. Only one thread can execute a method at a time in the monitor. A thread enters the monitor by acquiring a lock on the monitor and exits by releasing the lock. *Any object can be a monitor*. An object becomes a monitor once a thread locks it. Locking is implemented using the synchronized keyword on a method or a block. A thread must acquire a lock before executing a synchronized method or block. A thread can wait in a monitor if the condition is not right for it to continue executing in the monitor.

wait(), notify(), and notifyAll()

Use the wait(), notify(), and notifyAll() methods to facilitate communication among threads.

The wait(), notify(), and notifyAll() methods must be called in a synchronized method or a synchronized block on the calling object of these methods. Otherwise, an IllegalMonitorStateException would occur.

The wait() method lets the thread wait until some condition occurs. When it occurs, you can use the notify() or notifyAll() methods to notify the waiting threads to resume normal execution. The notifyAll() method wakes up all waiting threads, while notify() picks up only one thread from a waiting queue.

Example: Using Monitor

Task 1

```
synchronized (anObject) {  
    try {  
        // Wait for the condition to become true  
        while (!condition)  
            anObject.wait();  
        resume  
        // Do something when condition is true  
    }  
    catch (InterruptedException ex) {  
        ex.printStackTrace();  
    }  
}
```

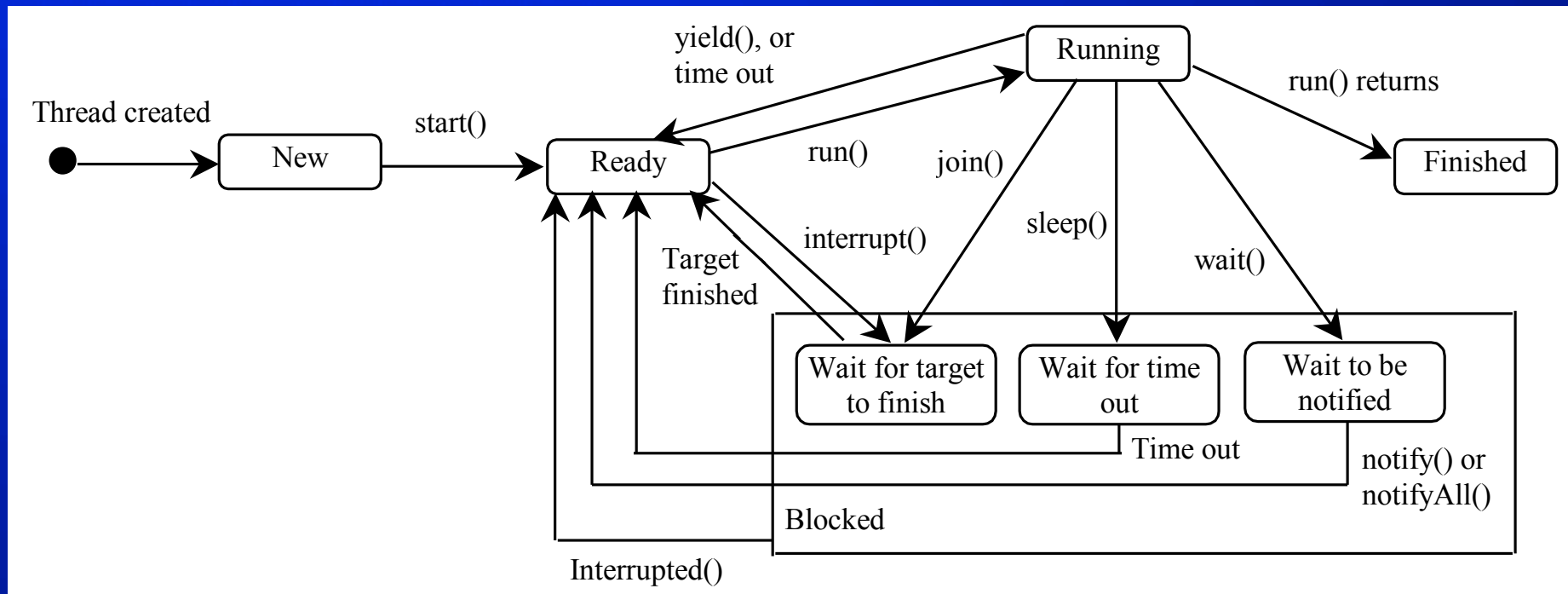
Task 2

```
synchronized (anObject) {  
    // When condition becomes true  
    anObject.notify(); or anObject.notifyAll();  
    ...  
}
```

- ✦ The wait(), notify(), and notifyAll() methods must be called in a synchronized method or a synchronized block on the receiving object of these methods. Otherwise, an IllegalMonitorStateException will occur.
- ✦ When wait() is invoked, it pauses the thread and simultaneously releases the lock on the object. When the thread is restarted after being notified, the lock is automatically reacquired.
- ✦ The wait(), notify(), and notifyAll() methods on an object are analogous to the await(), signal(), and signalAll() methods on a condition.

Thread States

A thread can be in one of five states:
New, Ready, Running, Blocked, or
Finished.



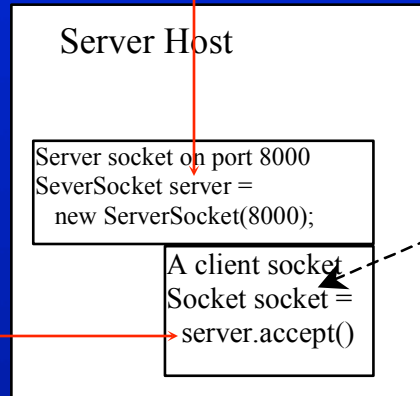
Networking (TCP)

Client/Server Communications

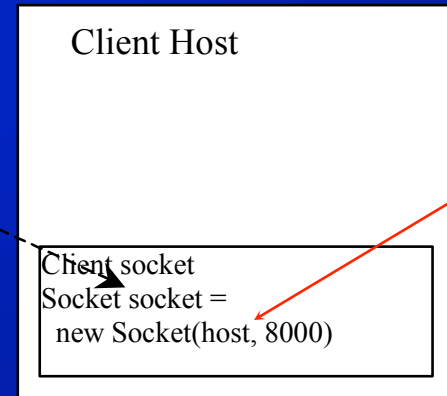
The server must be running when a client starts. The server waits for a connection request from a client. To establish a server, you need to create a server socket and attach it to a port, which is where the server listens for connections.

After the server accepts the connection, communication between server and client is conducted the same as for I/O streams.

After a server socket is created, the server can use this statement to listen for connections.

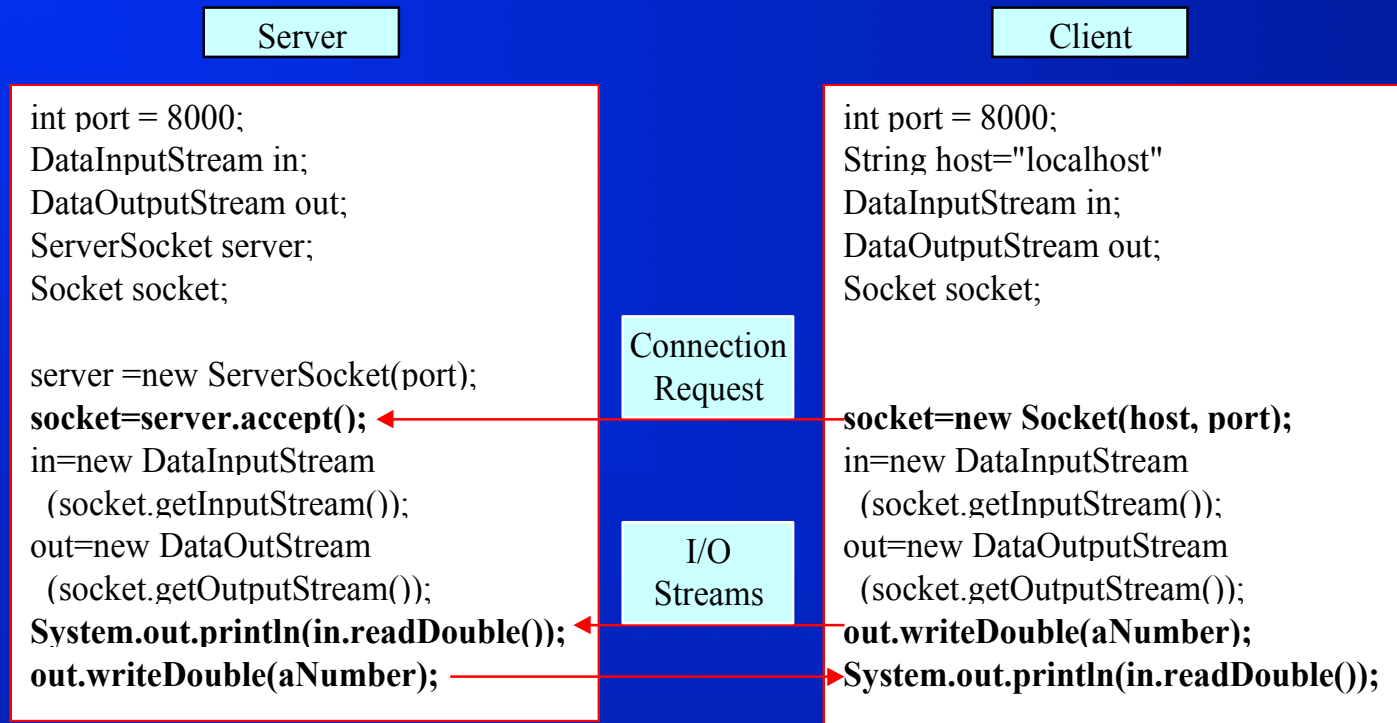


I/O Stream



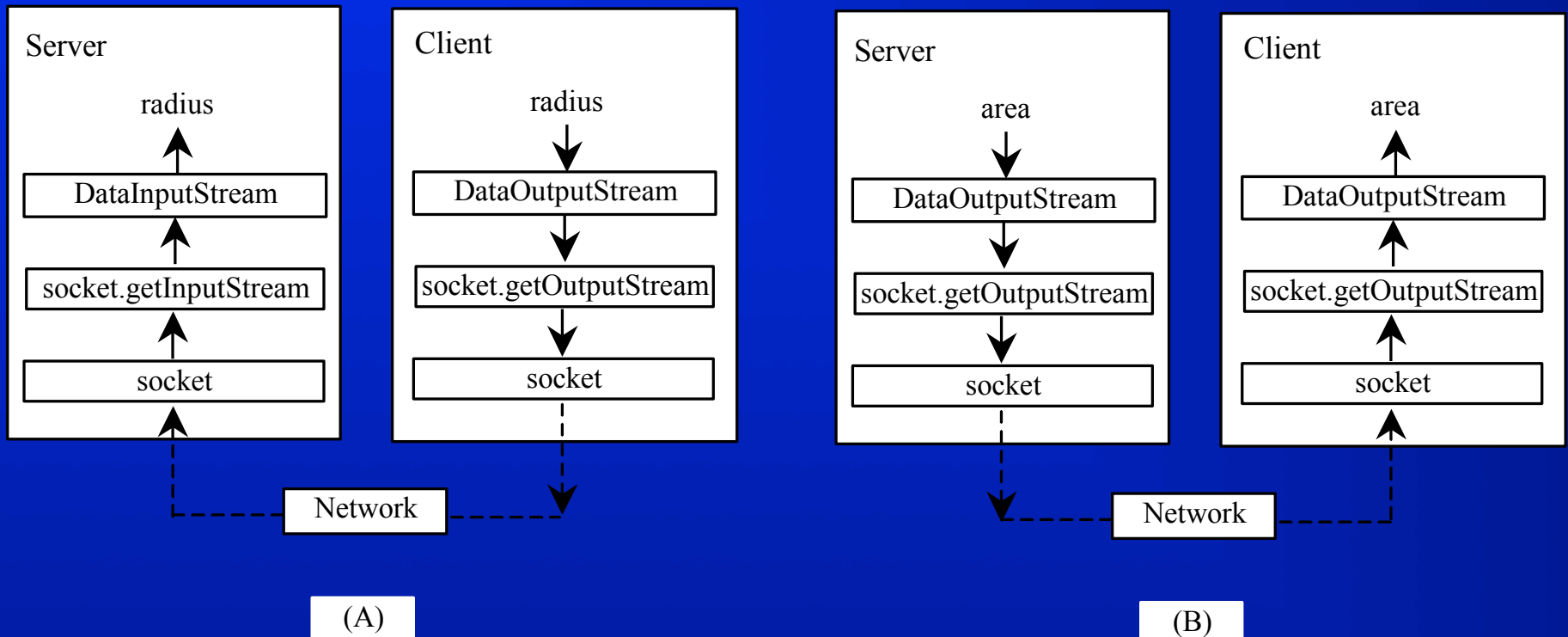
The client issues this statement to request a connection to a server.

Data Transmission through Sockets



```
InputStream input = socket.getInputStream();
OutputStream output = socket.getOutputStream();
```

A Client/Server Example



The InetAddress Class

Occasionally, you would like to know who is connecting to the server. You can use the InetAddress class to find the client's host name and IP address. The InetAddress class models an IP address. You can use the statement shown below to create an instance of InetAddress for the client on a socket.

```
InetAddress inetAddress = socket.getInetAddress();
```

Next, you can display the client's host name and IP address, as follows:

```
System.out.println("Client's host name is " +  
    inetAddress.getHostName());  
System.out.println("Client's IP Address is " +  
    inetAddress.getHostAddress());
```

Serving Multiple Clients

Multiple clients are quite often connected to a single server at the same time. Typically, a server runs constantly on a server computer, and clients from all over the Internet may want to connect to it. You can use threads to handle the server's multiple clients simultaneously. Simply create a thread for each connection. Here is how the server handles the establishment of a connection:

```
while (true) {  
    Socket socket = serverSocket.accept();  
    Thread thread = new ThreadClass(socket);  
    thread.start();  
}
```

The server socket can have many connections. Each iteration of the while loop creates a new connection. Whenever a connection is established, a new thread is created to handle communication between the server and the new client; and this allows multiple connections to run at the same time.

Good lucks in final!